ThProg — Fold in Haskell

8. Oktober 2018

```haskell
import qualified Prelude

-- Paare
fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y

-- Funktionskomposition
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)

-- Identitätsfunktion
id :: a -> a
id x = x

data Bool = True | False deriving (Prelude.Show, Prelude.Eq)

foldb :: a -> a -> Bool -> a
foldb a b True  = a
foldb a b False = b

data Nat = Zero | Succ Nat deriving (Prelude.Eq)

foldn :: a -> (a -> a) -> Nat -> a
foldn c g Zero     = c
foldn c g (Succ n) = g (foldn c g n)

zero = Zero
one = Succ Zero
two = Succ one
three = Succ two
four = Succ three

add :: Nat -> Nat -> Nat
add  m = foldn m Succ

mult :: Nat -> Nat -> Nat
```

```haskell
mult m = foldn Zero (add m)

exp :: Nat -> Nat -> Nat
exp m = foldn (Succ Zero) (mult m)

data List a = Nil | Cons a (List a) deriving (Prelude.Show, Prelude.Eq)

-- fold für Listen
foldL :: b -> (a -> b -> b) -> List a -> b
foldL c g Nil = c
foldL c g (Cons x xs) = g x (foldL c g xs)

-- scan für Listen
scanL :: b -> (a -> b -> b) -> List a -> List b
scanL c g Nil = Cons c Nil
scanL c g (Cons x xs) = Cons (g x y) ys
  where ys@(Cons y _) = scanL c g xs

-- alternativ ohne @-Syntax:
scanL' :: b -> (a -> b -> b) -> List a -> List b
scanL' c g xs = snd (scanLHelper c g xs)
  where
    scanLHelper c g Nil = (c, Cons c Nil)
    scanLHelper c g (Cons x xs) = (z, Cons z (snd ys))
      where
        ys = scanLHelper c g xs
        z = g x (fst ys)

length :: List a -> Nat
length l = foldL Zero (\x -> Succ) l

snoc :: a -> List a -> List a
snoc a l = foldL (Cons a Nil) Cons l

reverse :: List a -> List a
reverse l = foldL Nil snoc l


-- cat ohne foldr
concat :: List a -> List a -> List a
concat Nil = id
concat (Cons x xs) = \ys -> Cons x (concat xs ys)

-- cat mit foldr
cat' :: List a -> List a -> List a
cat' xs ys = foldL ys Cons xs

data Tree a = Leaf a | Node (Tree a) (Tree a) deriving (Prelude.Show, Prelude.Eq)

-- fold für Bäume
```

```haskell
foldt :: (a -> b) -> (b -> b -> b) -> Tree a -> b
foldt c g (Leaf x) = c x
foldt c g (Node x y) = g (foldt c g x) (foldt c g y)

-- front ohne foldt
front :: Tree a -> List a
front (Leaf x) = Cons x Nil
front (Node x y) = concat (front x) (front y)

-- front mit foldt
front' :: Tree a -> List a
front' = foldt (\x -> Cons x Nil) concat

-- Fakultätsfunktion aus der Vorlesung
fact :: Nat -> Nat
fact = f . foldn c h
  where
    f = snd
    c = (Zero, one)
    h = (\(x,y) -> (Succ x, mult (Succ x) y))
```